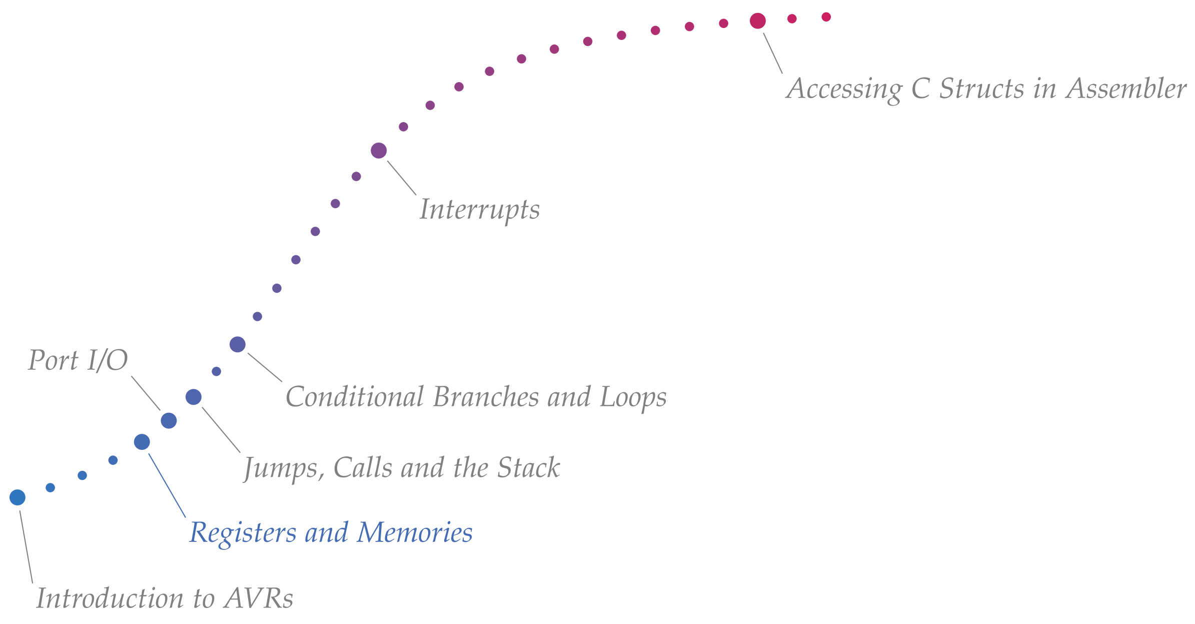
www.avrbeginners.net

General/Assembler Tutorial

Registers and Memories

Author: Christoph Redecker
Version: 1.0.2



Accessing C Structs in Assembler
Interrupts
Port I/O
Conditional Branches and Loops
Jumps, Calls and the Stack
Registers and Memories
Introduction to AVRs

AVRs have a number of different registers and memories, such as the general purpose registers, program memory and the I/O registers. This tutorial explains their purpose, usage and limitations. The EEPROM is *not* dealt with here.

The most important registers are the general purpose registers; the most important memory is the program memory — these are explained first, as they are responsible for the main program flow.

General Purpose Registers

The most important registers are the *general purpose registers*. They are used for arithmetic operations like adding and subtracting numbers, for comparing numbers and for more arcane things like indexing and setting jump destinations.

Read more about branches and jumps in the "Conditional Branches and Jumps" Tutorial.

Having 32 registers for working with numbers is quite convenient If you're only writing small programs, you might even get away without using additional storage space like the SRAM. In assembler, the registers are simply called r0 ... r31. You can also give them a more meaningful name with the `.def` directive and use that later:

The `.def` directive is used to assign a symbol (a name) to a register.

```
.def counter=r16
...
inc counter // same as inc r16
```

The double slash (`//`) starts a comment.

The name doesn't have to be *more* meaningful:

```
.def temp=r16
...
```

Assigning a name to a register is usually done for registers which serve a particular purpose in your program. If you are using interrupts, you will need a registers for storing the AVR's status register. Having a register with the value zero is (quite often) also a good thing:

There's a tutorial on interrupts as well.

```
.def zeroreg=r2
.def sregsave=r3
```

Why use r2 and r3 for that, and not, say, r16 and r17? Not all general purpose registers have the same features, and it is good practice to use registers which don't have more features than necessary for their specific purpose. The relevant features are:

- the usage as a "result register" of certain instructions (applies to r0 and r1),
- the ability to use immediate values (applies to r16 ... r31),
- the ability to use 16–bit operations (applies to the register pairs starting at r24),

- the ability to use the register as part of a pointer (applies to the register pairs starting at R26, which are X, Y and Z),
- the ability to use the register for operations on the program memory (applies to the register pair Z only).

Result Registers R0 and R1

Some instructions leave a result in R0, some in R0 and R1. One of these is `lpm`, if no destination register is specified. In that case, R0 is the implied destination:

```
ldi ZL, low(2*somewhereInFlash)
ldi ZH, high(2*somewhereInFlash)
lpm // now r0 contains the value at 2*somewhereInFlash
```

There is a reason why `somewhereInFlash` is multiplied by two, which will be discussed in the Flash section of this tutorial.

Unfortunately, there are two things in this piece of code that haven't been dealt with yet: the Z register pair and the program memory. They will beexplained later in this tutorial.

A class of instructions that leave a result in R0 and R1 are the multiply instructions:

The multiply instructions are `mul`, `muls`, `mulsu`, `fmul`, `fmuls` and `fmulsu`.

```
ldi r16, 35 // r16 = 35
ldi r17, 72 // r17 = 72
mul r16, r17 // now r0 = 0xD8, r1= 0x09; 0x09D8 = 2520
```

These instructions need two registers for their result, because a multiplication of two 8–bit values is up to 16 bits wide.

Immediate Values

Instructions like `ldi` (*load immediate*) can use 8–bit constants directly, as in

These instructions are `adiw`, `andi`, `cpi`, `ldi`, `ori`, `sbci`, `sbiw` and `subi`. `adiw` and `sbiw` are 16–bit operations and even more limited, see below!

```
ldi r16, 35 // r16 = 35
```

These instructions can not be used with all registers — their usage is limited to registers R16 ... R31. In the instruction set manual, this limitation is expressed (for example for `ldi`), in a line like

```
LDI Rd,K        16 <= d <= 31, 0 <= K <= 255
```

Here, the destination register is called `Rd`, the constant is `K`. Their valid ranges are specified in the right part of the line. It is clear that only registers R16 and above can be used, and the constant can be any 8–bit value.

The reason for this limitation is that the opcodes for these instructions simply do not have more space for the register number.

16–Bit Operations

Some *register pairs* can be used for 16–bit operations. These can only be used with the register pairs R24:R25 (has no name), R26:R27 (X), R28:R29 (Y) and R30:R31 (Z). An example with `adiw` (*add immediate to word*):

The 16–bit operations are `adiw`, `sbiw` and `movw`.

```
ldi XL, low(0x01FF) // XL = 0xFF
ldi XH, high(0x01FF) // XH = 0x01
adiw XL, 2 // result: X = 0x0201
```

The assembler automatically recognizes register names like XL and XH (same for Y and Z), there is no need to `.def` them.

Index Register Pairs (Pointers)

The register pairs X, Y and Z can be used as index registers. They are loaded with an address in the AVR's Flash, I/O or SRAM space, and the value that is stored at this address can be read or written. This is pretty much the same thing as a pointer in C. There are a variety of different ways to use these register pairs using `ld` (*load indirect from data space*):

```
ldi XL, low(somewhereInSram)
ldi XH, high(somewhereInSram)
ld r16, X // r16 = what is stored at somewhereInSram
ld r17, X+ // r17 = *X (C-like notation now)
           // X is incremented afterwards
ld r18, -X // X is decremented first
           // then r18 = *X
```

These instructions load a byte from the location pointed to by X, and can optionally post–increment or pre–decrement X (same for Y and Z). If you have structures in SRAM, the "displacement" variant of ld, ldd (*load indirect from data space with displacement*) can be used:

The `.equ` directive is used to assign a value to a label, much like `#define` in C.

```
.equ offset_to_someMember_in_myStruct = 3
ldi XL, low(myStruct)
ldi XH, high(myStruct)
ld r16, X+offset_to_someMember_in_myStruct
```

If you have defined all structure offsets somewhere (using `.equ`), you can access the structure's members quite conveniently. The value of the index register pair itself will remain unchanged.

It is also possible to store values using the index register pairs, using `st` (*store indirect to data space*) and `std` (*store indirect to data space with displacement*). An example:

```
ldi YL, low(somewhereInSram)
ldi YH, high(somewhereInSram)
ldi r16, 9
st Y+, r16 // now *somewhereInSram = 9
```

The same increment and decrement modifiers that were available for `ld` are also available for `st`, and `std` works as you would expect it.

The Z Register Pair: Accessing Program Memory (Flash)

Only the Z register pair can be used to access the program memory (Flash). This is done using the `lpm` (*load program memory*) and `spm` (*s*tore program memory) instructions. Only `lpm` is covered here, because the usage of `spm` is much more complex. `lpm` works like `ld`:

```
ldi ZL, low(2*somewhereInFlash)
ldi ZH, high(2*somewhereInFlash)
lpm r16, Z+ // r16 = what is stored 2*somewhereInFlash
lpm // r0 = what is stored at (2*somewhereInFlash)+1
```

The Z register pair can be used to access both SRAM and Flash — the choice is in the instruction: `ld` accesses the SRAM and `lpm` accesses the Flash.

Program Memory (Flash)

Instructions (opcodes) are stored in the AVR's program memory, which is a Flash memory. It can be programmed using a programmer, and hardware interfaces like *In–Circuit Serial Programming* (ISP) are provided for that purpose. Its endurance is limited to some 10 000 write/erase cycles (usually the memory will last longer). It can not only be accessed with a programmer, but also in your application code, using the `lpm` and `spm` instructions (`lpm` was used before in this tutorial).

So–called *Boot Loaders* use `spm` to store a program in Flash, without dedicated programming hardware.

The Flash is organised in *words* of 2 bytes, because the smallest instructions are 1 word (or 2 bytes) long. It starts at address $0x0000$, and when a program is assembled, it is implicitly placed at that address. It is also possible to deliberately place code (or constant data, we'll come to that later) at a chosen address, using the `.org` directive:

The `.cseg` (code segment) directive tells the assembler that the following items have to be placed in program memory. The `.org` directive sets the program address to a specific value.

```
.cseg
.org 0x10
nop // does nothing, but that's not the point
```

It is possible to place constant data in program memory, using the `.db` directive. These can also be labeled:

```
.cseg
somewhereInFlash:
```

```
.db "This is a string"
```

Now the label `somewhereInFlash` can be used to obtain the *word* address of the string that is stored at that address. In order to use this address with `lpm`, it must be converted to a *byte* address, by multiplication by two:

Why Flash addresses are somtimes multiplied by two

```
ldi XL, low(2*somewhereInFlash)
ldi XH, high(2*somewhereInFlash)
lpm r16, Z+ // now r16 = 'T'
```

Also note that, if you are coming from C, string constants are not automatically terminated with a zero. This has to be done manually, if desired:

When using `.db` to place an odd number of bytes in Flash, the last word will be padded with a zero. The next instruction will then start at a proper word boundary again.

```
.db "This is a C-string",0
```

Character constants can be enclosed in `''`s:

```
.db 'a','b','\n'
```

SRAM

The Flash should not be used for storing variables due to its write/erase limitation. However, if the 32 general purpose registers are not enough, variable data can be stored in SRAM. The SRAM is placed above the I/O and extended I/O registers (explained in the next sections), and its first addres depends on the number of extended I/O registers, which varies from device to device. Fortunately, we don't need to worry about these details — the assembler will know the relevant numbers if a device's include file is given, as in

```
#include <m168Pdef.inc>
```

It is not possible to define *constants* in SRAM, but it is possible to reserve space for variables:

```
.dseg
somewhereInSram:
.byte 20
```

This reserves 20 bytes at the address `somewhereInSram`. This address is automatically determined by the assembler, but can also be chosen with the `.org` (just like for the program memory). This reserved space can now be accessed with a number of different instructions. The simplest would be `lds` (*load direct from data space*) and `sts` (*store direct to data space*):

```
ldi r16, 5
sts somewhereInSram
lds r17, somewhereInSram+19
```

This piece of code writes the number 5 to the first reserved location at `somewhereInSram`, and loads the value at the last reserved location to R17.

The other possibilities of accessing the SRAM have already been discussed in the General Purpose Registers section.

Whenever you want to "work" with values stored in SRAM, you first need to load them into general purpose registers. Whenever you run out of general purpose registers, you need to store something in SRAM, essentially freeing a general purpose register.

I/O Registers

AVRs have internal peripherals, which are controlled through *I/O registers*. These registers start directly above the general purpose registers at address 0x20 (the last general purpose register is R31 at address 0x1F). They can be accessed with instructions like `ld`, `lds` or `sts`, but these instructions need 2 CPU cycles for execution. The AVR provides faster instructions, `in` (*load I/O location to register*) and `out` (*store register to I/O location*), which just need one cycle:

Note: `in` and `out` can only be used for I/O locations up to (and including) 0x3F. Above that, the "normal" SRAM instruction must be used.

```
#include <m168Pdef.inc>
...
ldi r16, 0
out TCNT0, r16 // reset Timer/Counter 0
```

Register addresses (like TCNT0) are defined in the include files delivered by Atmel.

The `in` and `out` instructions can *not* be used at locations 0x40 and higher — these are in the *extended I/O space*.

Extended I/O Registers

As the AVRs grew and more peripherals were added, `in` and `out` could not serve enough I/O locations. Additional I/O registers are placed above the I/O registers, but are treated like normal SRAM:

```
#include <m168Pdef.inc>
...
ldi r16, 0
sts TCNT2, r16 // reset Timer/Counter 2
```

The extended I/O addresses are also defined in the include files, and the address format is correct for instructions like `ld` and `st`.